INTERFACE-BASED HIERARCHY FOR SYNCHRONOUS DATA-FLOW GRAPHS

Jonathan Piat[1]*, Shuvra S. Bhattacharyya*[2]*, and Mickael Raulet*[1]

(1) IETR/INSA, UMR CNRS 6164
Image and Remote Sensing laboratory, F-35043 Rennes, France
email: {firstname.lastname@insa-rennes.fr}
(2)Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD, 20742, USA
email: {ssb@umd.edu}

ABSTRACT

Dataflow has proven to be an attractive computation model for programming digital signal processing (DSP) applications. A restricted version of dataflow, termed synchronous dataflow (SDF), offers strong compile-time predictability properties, but has limited expressive power. In this paper we propose a new type of hierarchy in the SDF domain allowing more expressivity while maintaining its predictability. This new hierarchy semantic is based on interfaces that fix the number of tokens consumed/produced by a hierarchical vertex in a manner that is independent or separate from the specified internal dataflow structure of the encapsulated subsystem. This interface-based hierarchy gives the application designer more flexibility in iterative construction of hierarchical representations, and experimentation with different optimization choices at different levels of the design hierarchy.

Index Terms— Data-Flow programming, SDF graph, Scheduling, Code Generation.

1. INTRODUCTION

Since applications such as video coding/decoding or digital communications with advanced features are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. Most tools such as PeaCE [1], SynDEx [2] or PREESM [3] use as an entry point a model of the application associated to a model of the architecture. Data flow model is indeed a natural representation for data-oriented applications since it represents data dependencies between the operations allowing to extract parallelism. In this model the application is described as a graph in which nodes represent computations and edges carry the stream of data-tokens between operations. The Synchronous Data Flow (SDF) model allows to specify the number of tokens produced/consumed on each outgoing/incoming edges for one firing of a node. Edges can also carry initialization tokens, called delay. That information allows to perform analysis on the graph to determine whether or not the graph is schedule-able, and if so to determine an execution order of the nodes and application's memory requirements.

In basic SDF representation, hierarchy is used either as a way to represent cluster of nodes in the SDF graph or as parameterized sub-system. The purpose of this paper is to describe a hierarchy type allowing the designer to describe subgraph in a classical top down approach. Relevant information can also be extracted from this representation in order to ease the graph scheduling and to lead to a better implementation.

Section 2 explains the data flow semantics and particularly the synchronous data flow graphs, section 3 presents the existing hierarchy types and section 4 the proposed hierarchy type. Section 5 uses the described hierarchy type to design an example and provides some results. Finally, section 5 highlights the future work and concludes this paper.

2. SYNCHRONOUS DATA FLOW GRAPH

The Synchronous Data Flow (SDF) graph [4] is used to simplify the application specification, by allowing the representation of the application behavior at a coarse grain. This data flow model represents operations of the application and specifies data dependencies between the operations.

A Synchronous Data Flow graph is a finite directed, weighted graph $G =< V, E, d, p, c >$ where :

- V is the set of nodes; each node represents a computation that operates on one or more input data streams and outputs one or more output data streams.
- $E \subseteq V \times V$ is the edge set, representing channels which carry data streams.
- $d : E \rightarrow N \cup \{0\}$ $(N = 1, 2, \ldots)$ is a function with $d(e)$ the number of initial tokens on an edge e.

- $p : E \rightarrow N$ is a function with $p(e)$ representing the number of data tokens produced at e's source to be carried by e.

- $c : E \rightarrow N$ is a function with $c(e)$ representing the number of data tokens consumed from e by e's sink node.

The topology matrix is the matrix of size $|E| \times |V|$, in which each row corresponds to an edge e in the graph and each column corresponds to a node v. Each coefficient (i, j) of the matrix is positive and equal to N if N tokens are produced by the j^{th} node on the i^{th} edge. (i, j) coefficients are negative and equal to N if N tokens are consumed by the j^{th} node on the i^{th} edge. It was proved in [4] that a static schedule for graph G can be computed only if its topology matrix's rank is one less than the number of nodes in G. This necessary condition means that there is a Basic Repetition Vector (BRV) q of size $|V|$ in which each coefficient is the repetition factor for the j^{th} vertex of the graph. SDF graph representation allows use of hierarchy, meaning that for $v = G$, a vertex may be described as a graph. A vertex with no hierarchy is called an actor.

2.1. SDF to DAG translation

One common way to schedule SDF graphs onto multiple processors is to first convert the SDF graph into a precedence graph such that each vertex in the precedence graph corresponds to a single execution of an actor from the SDF graph. Thus each SDF graph actor A is "expanded into" q_A separate precedence graph vertices, where q_A is the component of the BRV that corresponds to A. In general, the SDF graph aims at exposing the potential parallelism of the algorithm; the precedence graph may reveal more functional parallelism, moreover it exposes the available data-parallelism. A valid precedence graph contains no cycle and is called DAG (Directed Acyclic Graph). Unfortunately, the graph expansion due to the repetition count of each SDF node can lead to an exponential growth of nodes in the DAG. Thus, precedence-graph-based multiprocessor scheduling techniques, such as those developed in [5] [6], in general have complexity that is not polynomially bounded in the size of the input SDF graph, and can result in prohibitively long scheduling times for certain kinds of graphs (e.g., see [7]).

3. HIERARCHY TYPES IN SDF GRAPH

3.1. Repetition-based SDF hierarchy

Hierarchy has been described in [7], as a mean of representing cluster of actor in a SDF graph. In [7] clustering is used as a pre-pass for the scheduling described in [8] that reduces the number of vertices in the DAG, minimizing synchronization overhead for multi-threaded implementation and maximizing the throughput by grouping buffers [8]. Given a consistent SDF graph, this approach first clusters strongly connected components to generate an acyclic graph. A set of clustering techniques are then applied based on topological and data flow properties, to maximize throughput and minimize synchronization between clusters. This approach is a bottom-up approach, meaning that the starting point is a SDF graph with no hierarchy and it automatically outputs a hierarchical (clustered) graph. In order to ensure that clustering an actor may not cause the application to be deadlock, the authors (in [7]) describe five composition rules based on the data flow properties (Figure 1).

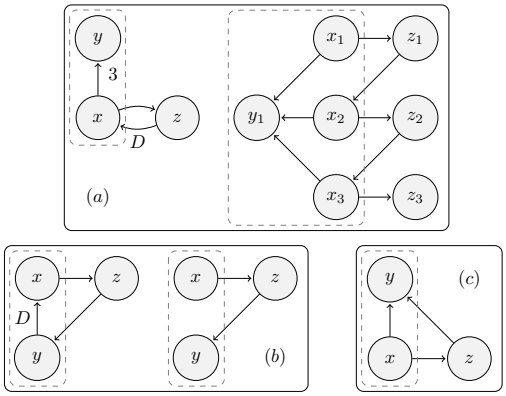

Fig. 1. ***(a)*** illustrate the violation of the first precedence shift condition, ***(b)*** illustrate the violation of the hidden delay condition, and ***(c)*** illustrate the violation of the cycle introduction condition

3.2. Parameter-based SDF hierarchy

Parameter-based SDF hierarchy has been introduced in [9] where the authors introduce a new SDF model called *Parameterized SDF*. This model aims at increasing SDF expressivity while maintaining its compile time predictability properties. In this model a sub-system (sub-graph) behavior can be controlled by a set of parameters that can be configured dynamically. These parameters can either configure sub-system interface behavior by modifying production/consumption rate on interfaces, or configure behavior by passing parameters (values) to the sub-system actors. In this model each sub-system is composed by three graphs: the init graph ϕ_i, the sub-init graph ϕ_s, the body graph ϕ_b.

Each activation of the sub-system, is composed by an invocation of ϕ_s followed by an invocation of ϕ_b. The init graph is effectively decoupled from the dataflow specification of the parent graph and invoked once, at the beginning of each (minimal periodic) invocation (see [9]). The sub-init graph performs reconfiguration that does not affect sub-system interface behavior and is activated more frequently than the init-graph which can modify sub-system interface behavior. In

order to maintain predictability, actors of ϕ_b are assigned a configuration which specifies parameters values. This value can either be a domain which specifies the set of valid parameter value combinations for the actor, or left unspecified, meaning that this parameter value will be determined at runtime.

4. INTERFACE-BASED SDF HIERARCHY

While designing an application, user might want to use hierarchy in a way to design independent graphs that can be instantiated in any design. From a programmer view it behaves as closures since it defines limits for a portion of an application. This kind of hierarchy must ensure that while a graph is instantiated, its behavior might not be modified by its parent graph, and that its behavior might not introduce deadlock in its parent graph. The rules defined in the composition rules ensure the graph to be deadlock free when verified, but are used to analyze a graph with no hierarchy. In order to allow the user to hierarchically design a graph, this hierarchy semantic must ensure that the composed graph will have no deadlock if every level of hierarchy is independently deadlock free. To ensure this rule we must integrate special nodes in the model that restrict the hierarchy semantic. In the following a hierarchical vertex will refer to a vertex which embeds a hierarchy level, and a sub-graph will refer to the graph representing this hierarchy level.

4.1. Special nodes

Source node: A Source node is a bounded source of tokens which represents the tokens available for an iteration of the sub-graph. This node behaves as an interface to the outside world. A source port is defined by the four following rules:

A-1 Source production homogeneity: A source node $Source$ produces the same amount of tokens on all its outgoing connections $p(e) = n \quad \forall e \in \{Source(e) = Source\}$.

A-2 Interface Scope: The source node remains write-locked during an iteration of the sub-graph. This means that the interface cannot be filled by the outside world during the sub-graph execution.

A-3 Interface boundedness: A source node cannot be repeated, thus any node consuming more tokens than made available by the node will consume the same tokens multiple times (ring buffer). $c(e)\%p(e) = 0 \quad \forall e \in \{source(e) = source\}$.

A-4 SDF consistency: All the tokens made available by a source node must be consumed during an iteration of the sub-graph.

Sink node: A sink node is a bounded sink of tokens that represent the tokens to be produced by an iteration of the graph. This node behaves as an interface to the outside world. A sink node is defined by the four following rules:

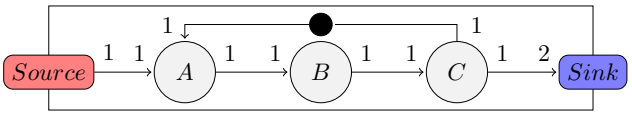

Fig. 2. Design of a sub-graph

B-1 Sink producer uniqueness: A sink node $Sink$ only has one incoming connection.

B-2 Interface Scope: The sink node remains read-locked during an iteration of the sub-graph. This means that the interface cannot be read by the outside world during the sub-graph execution.

B-3 Interface boundedness: A sink node cannot be repeated, thus any node producing more tokens than needed by the node will write the same tokens multiple times (ring buffer). $p(e)\%c(e) = 0 \quad \forall e \in \{target(e) = Sink\}$.

B-4 SDF consistency: All the token consumed by a sink node must be produced during an iteration of the sub-graph.

4.2. Hierarchy deadlock-freeness

Considering a consistent connected SDF graph $G = \{g, z\}$, $g = \{Source, x, y, Sink\}$ with $Source$ being a source node and $Sink$ being a sink node, and z being an actor. In the following we show how the hierarchy rules described above ensure the hierarchical vertex g to not introduce deadlocks in the graph G:

- if it exists a simple path going from x to y containing more than one arc, this path cannot introduce cycle since this path contains at least one interface, meaning that the cycle gets broken. User must take this into account to add delay to the top graph.

- Rules **A2-B2** ensure that all the data needed for an iteration of the sub-graph are available as soon as its execution starts, and that no external vertex can consume on the sink interface while the sub-graph is being executed. As a consequence no external vertex strongly connected with the hierarchical vertex can be executed concurrently. The interface ensures the sub-graph content to be independent to the outside world, as there is no edge $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ considering that $snk(\alpha') \notin \{x, y\}$) cannot happen.

- The designing approach of the hierarchy cannot lead to an hidden delay since even if a delay is in the sub-graph, an iteration of the sub-graph cannot start if its input interfaces are not full.

Those rules also guarantee that the edges of the sub-graph have a local scope, since the interfaces make the inner graph independent from the outside world. This means that when an edge in the sub-graph creates a cycle (and contains a delay), if the sub-graph needs to be repeated this iterating edge will not link multiple instances of the sub-graph.

The given rules are sufficient to ensure a sub-graph to not create deadlocks when instantiated in a larger graph.

4.3. Hierarchy scheduling

As explained in [10] interfaces to the outside world must not be taken into account to compute the schedule-ability of the graph. As in our hierarchy interpretation, interfaces have a meaning for the sub-graph, they must be taken into account to compute the schedule-ability, since we must ensure that all the tokens on the interfaces will be consumed/produced in an iteration of the sub-graph (see rules **A4-B4**).

Due to the interface nature, every connection coming/-going from/to an interface must be considered like a connection to an independent interface. Adding an edge e to graph G increases the rank of its topology matrix Γ if the row added to Γ is linearly independent from the other row. Adding an interface to a graph G composed of N vertices, and one edge e connecting this interface to G adds a linearly independent row to the topology matrix. This increases the rank of the topology matrix of one, but adding the interface's vertex will yield in a $N+1$ graph : $rank(\Gamma(G_N)) = N-1 \Rightarrow rank(\Gamma(G_{N+1})) = rank(\Gamma(G_N)) + 1 = (N+1) - 1$. The rank of the topology matrix remains equal to the number of vertices less one meaning that this graph remains schedule-able. Since adding an edge between a connected interface and any vertex of the graph, results in (in meaning) adding an edge between a newly created interface and the graph, it does not affect the schedule-ability considering the proof above. This means that a sub-graph can be considered schedule-able if its actor graph (excluding interfaces) is schedule-able.

Before scheduling a hierarchical graph, we must verify that every level of hierarchy is deadlock free. Applying the balance equation to every level is sufficient to prove the deadlock freeness of a level.

4.4. Hierarchy behavior

Interfaces behavior can vary due to the graph schedule. This behavior flexibility can ease the development process, but needs to be understood to avoid meaningless applications.

- Source behavior

As said in the source interface rules, a source interface can have multiple outgoing (independent) connection and reading more tokens than made available results in reading modulo the number of tokens available (circular buffer). This means that the interface can behave like a broadcast. In Figure 3, vertices A and B have to execute respectively 4 and 6 times to consume all the data made available by the port. In this example, the $Source$ interface will broadcast twice to vertex A and three times to vertex B. This means that the designer must keep in mind that the interfaces can have effect on the inner graph schedule.

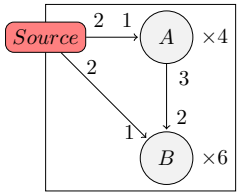

Fig. 3. Source example and its execution pattern

- Sink behavior

As said in the sink interface rules, a source interface can only have one incoming connection, and writing more tokens than needed on the interface results in writing modulo the number of tokens needed (circular buffer). In Figure 4, the vertex B writes 3 tokens in a $Sink$ that consumes only one token, due to the circular buffer behavior, only the last token written will be made available to the outside world. This behavior allows to easily design iterative pattern without increasing the number of edges. This behavior can also lead to mistakes (from the designer view) as if there is no precedence between multiple occurrences of a vertex that writes to an output port, a parallel execution of these occurrences leads to a concurrent access to the interface and as a consequence to indeterminate data in the sink node. This also leads to dead codes from the node occurrences that do not write to the sink node.

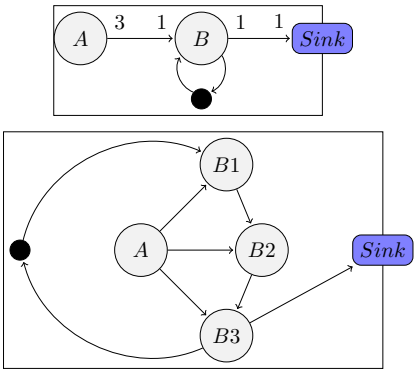

Fig. 4. Sink example and its precedence graph

4.5. Hierarchy improvements

As said earlier, this hierarchy type eases the designer work, since he/she can design subsystems independently and may instantiate them in any application. Not only easing the de-

signer work, this kind of hierarchy also improves the application with the same criteria than the clustering techniques. Those improvements are based on the designer's choice but it can be completed by automatic transformation allowing more performance to be extracted from the graph.

5. APPLICATION CASE STUDY

In this section we will show how the new hierarchy type (interface based hierarchy) can be used to design a IDCT2D_CLIP examples. The IDCT2D is a common application in image decoding which operates over a 8×8 matrix of coefficient to decode 8×8 pixel block. In the video decoding context the IDCT2D is followed by a clip operation which adds or not a sign bit on samples depending on the kind of prediction being used for the block (INTER or INTRA).

5.1. IDCT2D description

The IDCT2D_CLIP used in this example (Figure 5) is a recursive pattern using only 4 operations.

- *mux*: This actor, acts as a multiplexor. It outputs the data from the source port blockIn on its first firing and outputs the data from *trans* on the second firing.
- *idct*: Performs an IDCT on a vector of 8 elements.
- *trans*: Transposes an 8×8 matrix.
- *clip*: Apply an additional signed bit depending on the kind of the prediction type.

In this representation the trig operation is a null time operation which forces the loop to iterate twice. The IDCT2D_CLIP is defined using two level of hierarchy. The first level performs a classic IDCT2D by using IDCT1D and transposition of an 8×8 matrix. The additional level add the $clip$ operation which is specific to the video decoding algorithm. This operation computes on each sample a 9 bit signed integer for INTER prediction, while it does an 8 bit unsigned integer for INTRA prediction.

5.2. Structural analysis

The IDCT2D graph takes into account some of the specific behavior of the new hierarchy type. This graph is described as an entity consuming 64 tokens on its input port and producing 64 tokens on its output port. The trig operation forces the recursive pattern composed of *mux, idct, trans* to iterate twice in order to perform the idct onto line and column of the 8×8 block. The read operation being computed twice, it consumes twice the same 64 tokens on the input port of the graph, making it behaves in a manner analogous to a dynamic block of parameter values that is held fixed during an execution of the subsystem. The *trans* operation being computed twice writes two times 64 tokens on the output port of the graph. In this case the output port behaves as a circular buffer since only the last 64 tokens written will be made available to the outside world.

This kind of execution could not be described using the basic hierarchy as every produced token must be consumed and every vertex firing requires the number of tokens consumed to be available. In contrast, our new concept of interface-based hierarchy provides a description format that captures the desired behavior in a natural and efficient manner.

```
void idct2d_clip(int * blocki,
                 int * sign,
                 int * blocko){
  int block_out[64];
  idct2d(blocki, block_out);
  clip(block_out,sign,blocko);
}
```

Fig. 6. Automatic code generation of the first hierarchy level

```
void idct2d_clip(int * blocki,
                 int *sign,
                 int * blocko){
  int block_out[64];
  int triggers[2];
  int idct1d_loop[64];
  init_delay(idct1d_loop,64/*init_size*/);
  trig(triggers);
  for(i = 0; i<2 ; i ++){
    int *trigger = &triggers[(i*1)%2];
    int rows_in[64];
    int rows_out[64];
    mux(blocki,idct1d_loop,trigger,rows_in);
    for(j = 0; j<8 ; j ++){
      int *row_out = &rows_out[(j*8)%64];
      int *row_in = &rows_in[(j*8)%64];
      idct1d(row_in,row_out);
    }
    trans(rows_out,block_out,idct1d_loop);
  }
  clip(block_out,sign,blocko);
}
```

Fig. 7. Automatic code generation of the second hierarchy level

6. FURTHER WORK AND CONCLUSION

This paper introduces a classification of hierarchy types in synchronous dataflow representations and introduces a new hierarchy type that involves the designer more in the application optimization process by allowing him/her to modify the application structural description. In order to extract efficient schedules from synchronous dataflow graphs, we need to develop special scheduling and optimization methods to exploit hierarchical structure in the given application. Scheduling techniques discussed in [9] could be exploited as a starting point. The new hierarchy type proposed in our paper allows the designer to perform optimization on the application at a

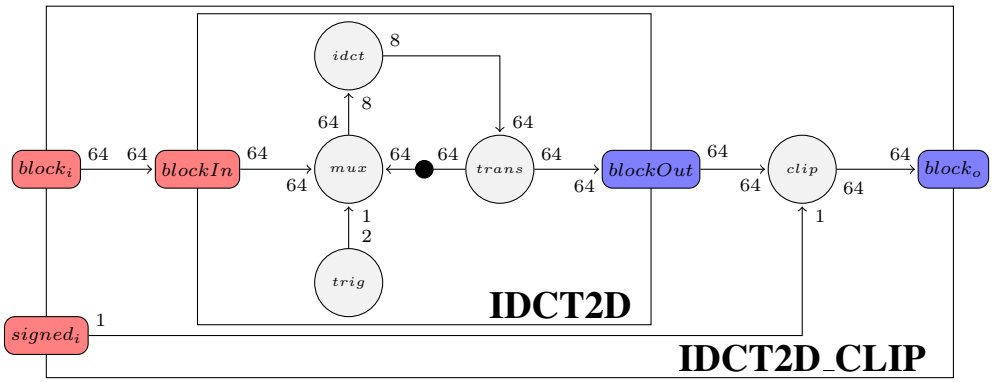

Fig. 5. IDCT2D_CLIP SDF graph designed with hierarchy type 2

structural level and provides a programming interface for hierarchical organization that is more natural in various contexts. In particular, our hierarchy representation is closer to C code semantics, and makes the application easier to describe for programmers who are for example more familiar with C, and less familiar with concepts such as repetitions vectors and subinit graphs. Our method allows reuse of graphs developed in other applications with no modifications, and offer more flexibility by allowing the description of execution patterns that do not map directly into conventional types of hierarchy. The Interface-based hierarchy for Synchronous Data-Flow Graphs has been implemented as the algorithm specification model in the tool PREESM [3]. A useful direction for further investigation is the development of techniques for optimized scheduling that are derived from our proposed new hierarchy type. Another useful direction for further investigation is the extension of the proposed interface-based hierarchy formulations to cyclo-static dataflow [11].

7. REFERENCES

[1] Wonyong Sung, Moonwook Oh, Chaeseok Im, and Soonhoi Ha, "Demonstration Of Codesign Workflow In PeaCE," in *in Proc. of International Conference of VLSI Circuit, Seoul, Koera*, 1997.

[2] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[3] Jonathan Piat, Mickaël Raulet, Maxime Pelcat, Pencheng Mu, and Olivier Déforges, "An extensible framework for fast prototyping of multiprocessor dataflow applications," in *IDT08: Proceedings of the 3rd International Design and Test Workshop*, Monastir, Tunisia, december 2008.

[4] E.A Lee and D.G Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, sept 1987.

[5] Harry William Printz, *Automatic mapping of large signal processing systems to a parallel machine*, Ph.D. thesis, Pittsburgh, PA, USA, 1991.

[6] E.A. Sih, G.C.; Lee, "Dynamic-level scheduling for heterogeneous processor networks," in *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, October 1990, pp. 42–49.

[7] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.

[8] C. Hsu, J. L. Pino, and S. S. Bhattacharyya, "Multithreaded simulation for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, Anaheim, California, June 2008, pp. 331–336.

[9] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

[10] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-Static Dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.